LID: Retry Relay Station and Fusion Shell

Julien Boucaron, Anthony Coadou, Robert de Simone

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

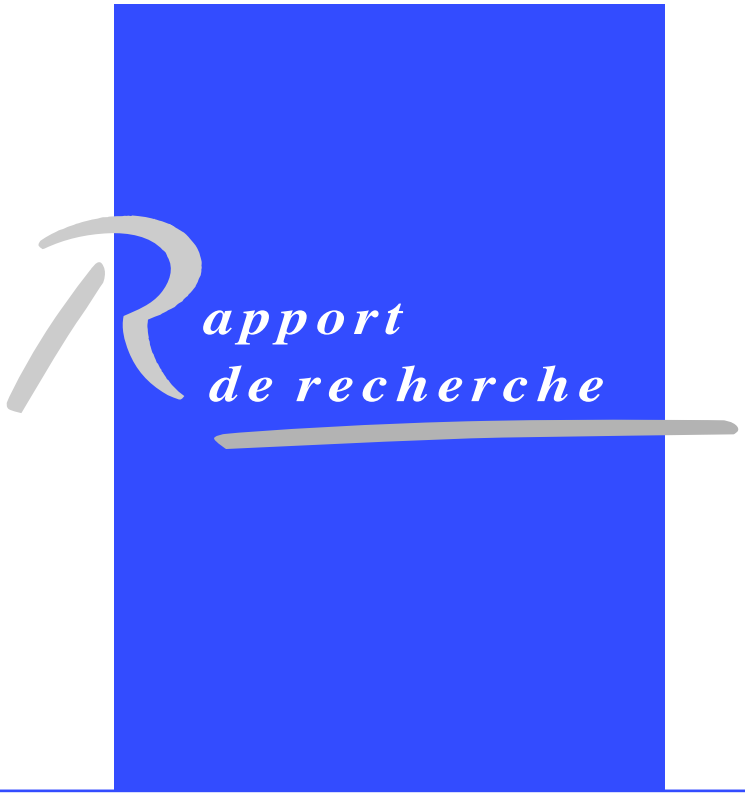
Latency-Insensitive Design: Retry Relay-Station and Fusion Shell

Julien Boucaron — Anthony Coadou — Robert de Simone

N° 7293

May 2010

Thème COM



ISSN 0249-6399 ISRN INRIA/RR--7293--FR+ENG

Latency-Insensitive Design: Retry Relay-Station and Fusion Shell

Julien Boucaron*, Anthony Coadou*, Robert de Simone

Thème COM — Systèmes communicants
Projet Aoste

Rapport de recherche n° 7293 — May 2010 — 14 pages

Abstract: This paper introduces a new variant implementation of Latency-Insensitive Design elements. It optimizes area footprint of so-called Shell-Wrappers being partially fused with their input Relay-Stations. The modified Relay-Station is called a Retry Relay-Station. We show correctness of this implementation and provide comparative results between a regular implementation and our new one on both FPGA and ASIC.

Key-words: Latency Insensitive Design, Implementation

FMGals 2009

This paper is electronically published in Electronic Notes In Theoretical Computer Science `http://www.elsevier.nl/locate/entcs`

* AOSTE, INRIA, Sophia-Antipolis, France

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Latency-Insensitive Design: Retry Relay-Station and Fusion Shell

Résumé : This paper introduces a new variant implementation of Latency-Insensitive Design elements. It optimizes area footprint of so-called Shell-Wrappers being partially fused with their input Relay-Stations. The modified Relay-Station is called a Retry Relay-Station. We show correctness of this implementation and provide comparative results between a regular implementation and our new one on both FPGA and ASIC.

Mots-clés : Latency Insensitive Design, Implementation

1 Introduction

Interconnects play a major role in high-performance circuits and systems. Latency-Insensitive Design (LID) was introduced by Carloni *et al.* [8] as a methodology to cope with multiple-clock-cycle latencies due to long interconnect wires. In their seminal paper, they establish behavior trace equivalence between the latency insensitive "implementation" and the synchronous specification. We can classify LID implementations in three classes: dynamic synchronous [5, 6, 7, 10, 12], static synchronous [4, 5, 11] and dynamic asynchronous [13]. Dynamic or static in this context stands for dynamically scheduled or statically scheduled respectively, whereas synchronous or asynchronous denotes the implementation style. In this paper, we focus only on dynamic synchronous compositional implementation (dynamic asynchronous compositional implementation can be built from a synchronous one, as in [2]).

The design-flow of LID starts from an ideal synchronous design, in which timing closure cannot be reached because IP blocks are too distant from each other.

It consists of several steps:

(1) Encapsulate each IP (called Pearl) into a Shell-Wrapper (SW),

(2) Divide each long wire into sections, through the addition of intermediate Relay-Stations (RS),

(3) Apply place-and-route,

(4) If timing closure cannot be reached, iterate from second step.

The core hypothesis of LID is that a Pearl is a synchronous IP that *can be clock-gated within a clock cycle*. A secondary hypothesis is that latencies are integers, which is natural in presence of a single clock. We segment each long wire of latency l with $l-1$ Relay-Stations, connected together with wires having unit latency. A Relay-Station acts as a "smart" signal repeater.

As already mentioned, a Shell-Wrapper encapsulates each Pearl. Its role is two-fold: it implements a part of the Latency Insensitive Protocol ensuring synchronization between data, and it drives the Pearl as a clock-gating mechanism exactly when all data have arrived.

Actually, the LID can be seen as a synchronous implementation of well-known asynchronous protocols [1]. The main difference with asynchronous design is that LID uses the classic synchronous design flow. It does not claim to solve the timing closure issue; careful floor-planning/partitioning and interconnect planning are mandatory.

The extra elements introduced in the Latency Insensitive interconnect structure allow to design the full system without affecting the original IPs, provided the single property of accepting gated clock inputs.

Contribution. We introduce a new implementation of LID, where the Shell-Wrapper input stage is optimized for area through a "fusion" with its specific input Relay-Stations. The resulting compound element is called Retry Relay-Station; it lets remove bypassable buffers in both control and data-flow parts of the new Fusion Shell. We show correctness of the implementation and discuss its main features and performance.

2 Regular Implementation

We describe briefly in this section a regular implementation of Latency-Insensitive Design. Most of dynamic implementations use bypassable buffers on the input stage of the Shell-Wrapper, as shown in figure 1. According to the clock-gating and the condition on the mux, the buffer samples the input – is bypassed by the wire – or sends its sampled input. In this paper, we suppose the use of flip-flops to store data, but implementations using transparent latches [12] would work in the same way.

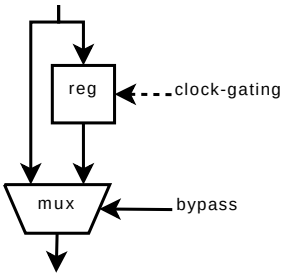


Figure 1: Bypassable register

2.1 Relay-Station

We do not fully detail the implementation of the Relay-Station as can be found in [6]. Figure 2 depicts only the control part of the Relay-Station, using a Mealy machine with three states (describing the use of the two internal registers of the data-path) and one optional state for errors. Actually, the *Error* state is used only for simulation and verification purposes, but is not necessary for implementation, as it will never be reached.

The Relay-Station (RS) is a two-place register, holding at most one initial data:

- *Empty* state: when the RS is empty, it waits for a valid input until having one; then, it catches the valid data, and goes to "Half" state.
- *Half* state does regular wire-pipelining (sampling valid input and sending it next clock cycle if no downward "stop"). If it receives a "stop" from a downward RS or SW, and at the same time a new data, then it has to hold both valid data and go to "Full" state.
- *Full* state sends "stop" upward. In "Full" state, it is necessary to halt the upward production of data, because there is a congestion in the system downward. This principle is called *back-pressure*. In addition, it should never receive a new data or it would be a design error.

The previous automaton is two-fold on the data-flow of the RS: it both drives the clock of *main* and *aux* registers, and also drives the mux.

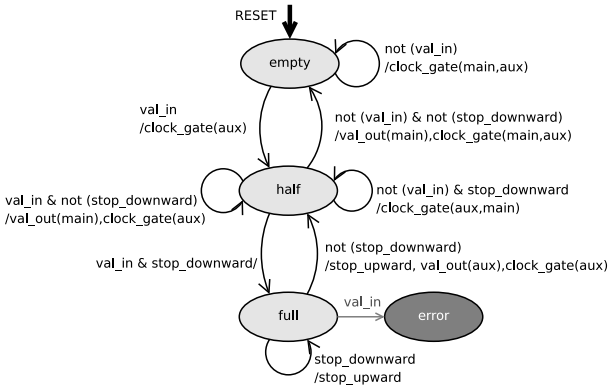


Figure 2: Mealy machine of a Relay-Station

Notice that the Relay-Station is order-preserving: the "aux" register is always bypassed, except if a RS, already holding a data in its "main" register, receive both a "stop" signal and a new data (from "Half" to "Full" states). In this case, the main is stored in the "aux" register while the new one is stored in the "main". Then it sends the data contained in "aux" to go back to "Half" state.

2.2 Shell-Wrapper

We now focus on the interactions between the input stage of the usual Shell-Wrapper and input/output Relay-Stations as shown in figure 3: we have a synchronous clock cycle in between each "down-half" input Relay-Station, passing through the Shell-Wrapper and the IP, until reaching each "up-half" output Relay-Station. The input-stage of the Shell-Wrapper has a bypassable buffer on each input channel. (If there is more than a clock cycle from input RSs to output RSs, then we need to have bypassable FIFOs having at least the same size as latency found from input to output RSs).

The Shell works as follows:

- The Shell enables the IP clock when there is one valid data on each input (or in each bypassing buffer) and no incoming stop from downward. When the clock is enabled, then as a result all valid outputs are produced.
- The bypassable buffer stores the input when both following conditions are satisfied: there must be an incoming data; the Shell is still waiting for at least one of its inputs *or* it receives a "stop" from downward.
- The Shell sends a stop upward if the bypassable buffer already holds a valid data and there is at least one incoming stop, or at least one missing input.

Many other implementations have been proposed [3, 7, 9], involving non-bypassable registers on inputs, as well as non-bypassable registers on outputs, or both. Actually, those

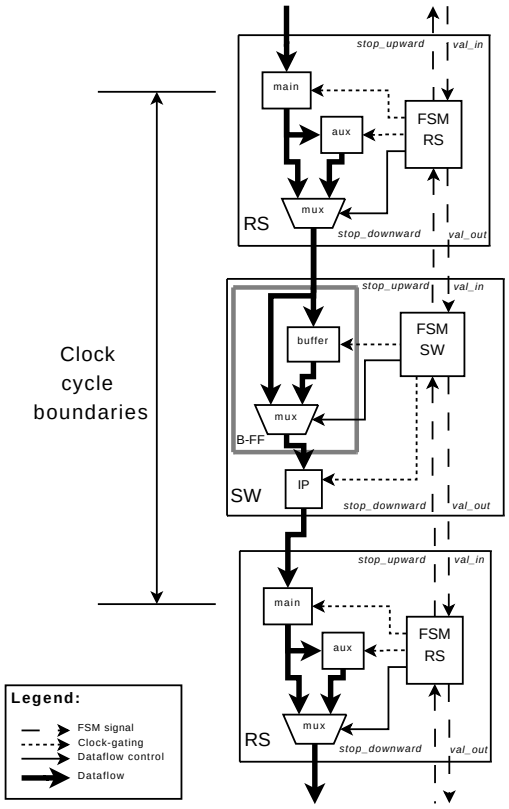


Figure 3: Shell-Wrapper and Relay-Station

solutions impose some buffering in the Shell, that can performed by input or output Relay-Stations as well.

3 Our Implementation

The basic idea of our implementation is to remove all bypassable buffers because valid data are present in "main" and "aux" buffers of input Relay-Stations, and also because synchronous signals can go from "down-half" Relay-Stations to "up-half" Relay-Stations. We state that the purpose of a Shell is to drive the execution of its pearl, while the intermediary storage is the responsibility of Relay-Stations. We slightly modify the regular implementation in order to remove as much as non-strictly necessary storage from the Shell, in order to give more "breathing space" in the placement of Relay-Stations and then, in some cases, to reduce global latencies. An example is given in figure 4.

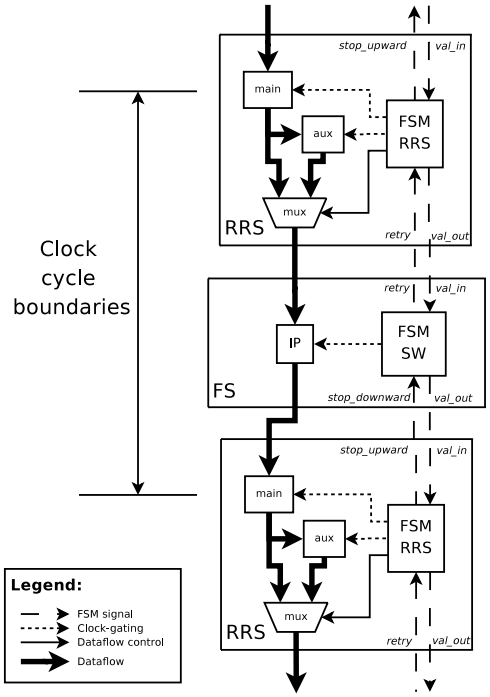


Figure 4: Fusion Shell and RRS

3.1 Retry Relay-Station

When a Relay-Station sends a data, the receiver is supposed to be ready, which is not the case in our fusion Shell which is strictly combinatorial. The trick here is to modify input Relay-Stations in order to keep the valid data and retry sending the data until the fusion Shell has consumed it. We call such Relay-Station a Retry Relay-Station (RRS).

We detail now the Retry Relay-Station: it has also two registers on both data and control path, as in a regular RS. A Mealy machine is shown in figure 5.

- When a valid data arrives, we put the valid data in "main" register and go to "Retry" state.
- In *Retry* state, whatever the input signal, we send the valid data. If both a retry and a new valid data is coming, then we go to the "Full" state while holding the new arriving data in the "main" register and send the old one to the "aux" register. We go back to "Empty" state when we do not receive a "retry".
- In *Full* state, whatever the input signal, we send a stop upwards and we send also the oldest data in the "aux" register. We go back to "Retry" state when we do not receive a "retry".

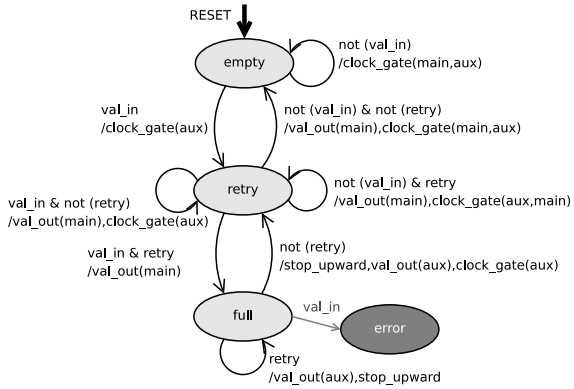


Figure 5: Mealy machine of a Retry Relay-Station

Moreover, the registers are clock-gated whenever they do not receive a new data. Basically, the behaviour is the same as the regular one, except the fact that it repeats the "val_out" signal until it has been acknowledged (absence of "retry"). Figure 6 gives an example of composition of controls of two successive RRS. It shows that the signals does not take more than a clock cycle to go from a Relay-Station to another.

3.2 Fusion Shell

The *Fusion-Shell* implementation is strictly combinatorial on both control and data. It executes the IP when there is no downward stop (the next RRSs are ready to receive the data) and all inputs are ready (the IP itself is ready to perform the computation). Otherwise, it sends a "retry" to upward RRSs which have sent valid data when there is either a missing input data or a downward stop. We assume that all inputs of the *fusion* Shell are only RRSs.

3.3 Correctness

We show the correctness of the previous implementation using a trace-equivalence (order preservation).

The Fusion-Shell is strictly combinatorial, hence order-preserving. It has already be shown that a Relay-Station is order-preserving and that the back-pressure protocol never loses or overwrites a data provided the following hypothesis: when the Relay-Station sends a "stop" upwards, it does not receive a new data.

Our RRS relies also on the same set of hypotheses. When a RRS is empty, it does not send anything; if it receives a data, it is stored in the "main" buffer and sent at the next clock cycles until there is no more "retry" signal coming from downward. If it holds a data and receives a new one, it uses the "aux" buffer to store the old data and put the new one in the "main" buffer, then at the next clock cycle it sends a "stop" upward. The previous

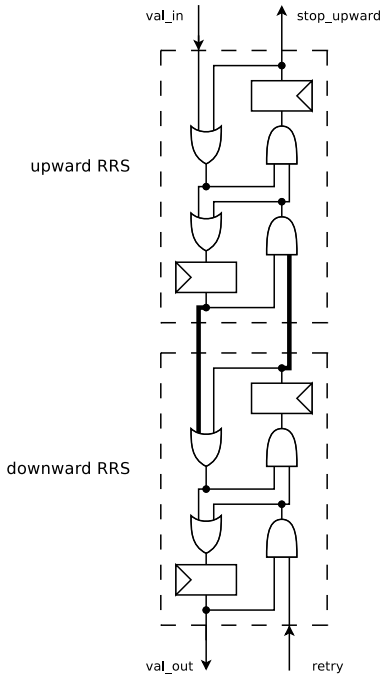


Figure 6: Control flows of two RRS.

hypothesis ensures that we do not receive another data. Finally, the RRS sends the oldest data, followed by the other one. The RRS is order-preserving.

4 Results

In this section we discuss the results for both FPGA and ASIC implementation given in tables 2, 3 and 4.

For the FPGA experiments, the verilog design was mapped using Xilinx ISE 10.1.02 onto both Spartan3-1000 and Virtex5-LX50. On each architecture, we tried both the area and speed optimization heuristics with high effort.

For the ASIC experiment, we used the FreePDK 45 nm version 1.2 from North Carolina State University and the Standard-Cell library from Oklahoma State University, with Synopsys DC version Y-2006.06-SP4. The clock frequency in this case is 1 GHz.

For each mounting, two cases were studied. The higher table contains the data relative to a parametric example to see how our implementation of *only* the control parts of SW and RS (FS and RRS resp.) scales with an increasing number of inputs and outputs for the SW (FS resp.), with the same number of RS (RRS resp.) connected to the input of the SW (FS

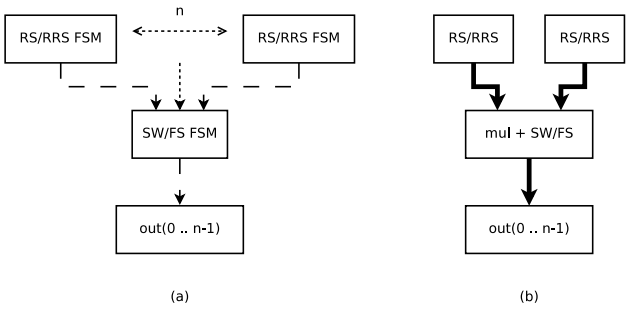


Figure 7: Two studied examples

RS (μm^2)	RRS (μm^2)	Area diff %
64.763399	61.008999	-5.79

Table 1: Area of control of RS and RRS, ASIC 45 nm, clock frequency 1 GHz

resp.), and a register of the same size on the output of the SW (FS resp.), as shown in figure 7a.

The lower table refers to an example of tiny data-path, to emphasize the area gain in favor of our new implementation. This design is composed of 2 RSs (2 RRSs resp.), a SW (FS resp.) and a integer multiplier (16x16 and 32x32) with both control and data-paths (figure 7b).

Area minimisation Area is of utter importance in implementation in VLSI. We are able to get an interesting area saving on the Shell-Wrapper on both data and control paths. That means a reduced overhead on each IP, and therefore allows a finer-grain design.

RS and RRS have the same data-path, as shown in table 1 control area is roughly the same, with a little advantage for the RRS in case of ASIC. In the case of FPGA, the area is not relevant due to its coarser granularity: the control takes exactly the same number of LUTs for RS and RRS.

- When considering only "pure" control part of our new implementation on both FPGA and ASIC, we are able to save area, while being able to sustain a faster clock rate. We can see on FPGAs that area gain is between 6 and 29 percent, and the clock rate gain is between 3 and 50 percent. Of course, those gains depend slightly on the quality of the mapping algorithm on the target architecture. We can see that the area gain in ASIC is in between 8.9 and 10.7 percent (there is about no slack on the biggest designs, needing a lot of buffers). Our new implementation scales in the same way as the regular one for the control part, with an interesting area save.
- When considering the simple design with the tiny data-path: the area save is slightly smaller within 4 to 6 percent on FPGA, because a lot of FFs present in LUTs are not

used in the mapping found by the FPGA placement-and-routing tool; the delay of the data-path dominates. When we look at the ASIC case, we find an area save between 16 to 10 percent, which corresponds roughly to the two removed 16/32bits bypassable buffers in the design.

Energy consumption Power minimization is also a mandatory objective in VLSI. We do not have applied any low-level power optimization on our designs.

For the FPGA implementations, the results are not relevant enough. They depend too widely on the architecture and heuristic used. For the ASIC implementation, we have an interesting power gain on both dynamic power with around 40 percent, and between 11 and 18 percent less on static power.

Clock frequency Our "control" part do not slow the clock frequency. On the contrary, it enables to raise significantly the clock frequency when mapped onto a FPGA.

The small dataflow example also shows that the overhead introduced by the LID is globally reduced by our implementation. On a Virtex5, the gain of frequency is between 10 to 20 percent. Notice that the results are strongly related to the place-and-route tool. For instance, on a Spartan3, a heuristic may lead to slightly worst results, while another will gain up to 10 percent.

5 Conclusions and discussion

This paper introduces a new implementation of Latency Insensitive Design, that let save area on Shell-Wrappers through a fusion of input Relay-Stations and the Shell. Regular Relay-Stations do not hold anymore a data once it has been sent.

The Retry Relay-Station sends its data (if present) until the Shell-Wrapper acknowledges reception that is coming after. We can then remove bypassing buffers on both control and data paths of the usual Shell-Wrapper and obtain the Fusion Shell-Wrapper. There is no additional area on both control and data-path in using Retry Relay-Stations versus regular Relay-Stations. We show the correctness of the implementation using trace-equivalence.

We provide detailed results on the implementation of both FPGA and ASIC. We show a gain in area, in clock rate in general, and also on both dynamic and static power.

One of the main problems in LID is to insert Relay-Stations to reach timing closure, as described in the introduction. Most of the time, high latencies are due to lot of wire congestions in the routing channels. Adding Relay-Stations in such areas is problematic and make the placement-and-routing tougher to solve.

Cortadella *et al.* [12] implementation is splitting Relay-Stations in two smaller parts. This helps the placement and routing, minimizes area, while having also time-borrowing due to the use of latches instead of flip-flops. Despite the fact we have implemented our Retry Relay-Stations using flip-flops, one can implement them using latches, with certainly additional gain on area, power and clock speed for ASICs.

In/Out	Control of **Regular Implementation**				Control of **Our Implementation**				**Gain**			
	Speed opt.		Area opt.		Speed opt.		Area opt.		Speed opt.		Area opt.	
	MHz	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*
2	240	24	237	20	246	17	250	17	+2.5%	-29%	+5%	-15%
4	169	50	136	39	246	17	250	17	+27%	-24%	+25%	-20%
8	148	87	114	81	167	70	146	64	+12%	-19%	+28%	-20%
16	128	153	129	157	182	120	182	120	+42%	-21%	+41%	-23%
32	110	311	117	312	166	235	167	235	+50%	-24%	+42%	-24%

In/Out	Multiplier with **Regular Impl.**					Multiplier with **Our Impl.**					**Gain**			
	Speed opt.		Area opt.			Speed opt.		Area opt.			Speed opt.		Area opt.	
	MHz	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*
16x16	81.1	135	74.6	87	134	78.6	100	82.2	81	100	-3%	-26%	+10%	-6%
32x32	54.0	234	49.8	229	262	52.8	228	54.1	223	196	-2%	-2%	+8%	-2%

Table 2: FPGA – Spartan3-1000, speed -4

In/Out	Control of **Regular Implementation**				Control of **Our Implementation**				**Gain**			
	Speed opt.		Area opt.		Speed opt.		Area opt.		Speed opt.		Area opt.	
	MHz	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*
2	664	18	508	17	905	16	720	15	+36%	-11%	+41%	-11%
4	557	35	437	33	639	32	494	31	+14%	-8%	+13%	-6%
8	439	73	360	64	532	62	430	59	+21%	-15%	+19%	-7%
16	313	128	330	127	409	117	376	116	+30%	-8%	+13%	-8%
32	354	284	292	252	365	231	313	230	+3%	-18%	+7%	-8%

In/Out	Multiplier with **Regular Impl.**					Multiplier with **Our Impl.**					**Gain**			
	Speed opt.		Area opt.			Speed opt.		Area opt.			Speed opt.		Area opt.	
	MHz	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*
16x16	192	84	190	82	134	229	78	229	78	100	+19%	-7%	+20%	-4%
32x32	105	148	104	146	262	116	142	116	142	196	+10%	-4%	+10%	-2%

Table 3: FPGA – Virtex5-LX50, speed -3

In/Out	Control of **Regular Implementation**	Control of **Our Implementation**	**Gain**
	Area (μm^2)	Area (μm^2)	*(%)*
2	209	186	-10.7
4	424	378	-10.7
8	818	740	-9.5
16	1653	1489	-9.9
32	3352	2994	-8.9

Mul	Multiplier with **Regular Impl.**			Multiplier with **Our Impl.**			**Gain**		
	Area	Power		Area	Power		Area	Power	
		dynamic	quiescent		dynamic	quiescent		dynamic	quiescent
	(μm^2)	*(mW)*	*(μW)*	(μm^2)	*(mW)*	*(μW)*	*(%)*	*(%)*	(%)
16x16	5715	1.99	31.6	4791	1.22	26.0	-16.1	-38.7	-17.8
32x32	17986	3.70	92.1	16215	2.20	81.4	-9.80	-40.6	-11.6

Table 4: ASIC 45 nm, clock frequency 1 GHz

References

[1] Kees Van Berkel. *Handshake Circuits: An Asynchronous Architecture for Vlsi Programming*. Cambridge University Press, 1994.

[2] Ivan Blunno, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Kelvin Lwin, and Christos Sotiriou. Handshake protocols for de-synchronization. In *in International Symposium on Advanced Research in Asynchronous Circuits and Systems. 2004*, pages 149–158. IEEE Computer Society Press, 2004.

[3] Pierre Bomel, Éric Martin, and Emmanuel Boutillon. Synchronization processor synthesis for latency insensitive systems. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*, pages 896–897, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Latency-insensitive design and central repetitive scheduling. In *MEMOCODE*, pages 175–183, 2006.

[5] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, 2007(1), 2007.

[6] Julien Boucaron, Jean-Vivien Millo, and Robert de Simone. Another glance at relay stations in latency-insensitive design. In *Electronic Notes in Theoretical Computer Science ENTCS 146-2*, pages 41–59, January 2006.

[7] Luca P. Carloni. The role of back-pressure in implementing latency-insensitive systems. In *Electronic Notes in Theoretical Computer Science ENTCS 146-2*, January 2006.

[8] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[9] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Coping with latency in soc design. *IEEE Micro*, 22(5):24–35, 2002.

[10] Mario R. Casu and Luca Macchiarulo. A Detailed Implementation of Latency Insensitive Protocols. In *FMGALS 2003 Proceedings*, 2003.

[11] Mario R. Casu and Luca Macchiarulo. A New Approach to Latency Insensitive Design. In *DAC'2004*, 2004.

[12] Jordi Cortadella, Michael Kishinevsky, and Bill Grundmann. Synthesis of synchronous elastic architectures. In *DAC*, pages 657–662, 2006.

[13] Christer Svensson. Synchronous Latency Insensitive Design. In *ASYNC'04*, 2004.

Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399